SPEARBIT

Gauntlet Security Review

Auditors

Gerard Persoon, Lead Security Researcher

Eric Wang, Lead Security Researcher

Emanuele Ricci, Security Researcher

Devansh Batham, Apprentice

Report prepared by: Pablo Misirov & Devansh Batham

June 13, 2022

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Gauntlet is the financial modeling and simulation platform for blockchains. Decentralized systems create new challenges for protocol developers, smart contract developers, and asset holders that you don't see in traditional development and asset management. Gauntlet uses tried and tested techniques from algorithmic trading to simulate crypto networks, inform design decisions, and drive successful design and participation.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Gauntlet's contract codebase according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 14 days in total, Gauntlet engaged with Spearbit to review Aera Contracts. In this period of time a total of 55 issues were found.

Summary

Project Name	Gauntlet
Repository	Aera Contracts
Commit	d48ddedf1dc70b9...
Type of Project	Treasury re-insurance, DeFi
Audit Timeline	May 2 - May 16, 2022
Methods	Manual Review

Issues Found

Critical Risk	2
High Risk	8
Medium Risk	8
Low Risk	9
Gas Optimizations	6
Informational	22
Total Issues	55

5 Findings

5.1 Critical Risk

5.1.1 Important Balancer fields can be overwritten by `EndTime`

Severity: *Critical Risk*

Context: ManagedPool.sol#L75-L77, ManagedPool.sol#L84-L86, LegacyBasePool.sol, WordCodec.sol

Description: Balancer's `ManagedPool` uses 32 bit values for `startTime` and `endTime` but it does not verify if those values exist within that range. Values are stored in a 32-byte `_miscData` slot in `BasePool` via the `insertUint32()` function. Nevertheless, this function does not strip any excess bits, resulting in other fields stored in `_miscData` to be overwritten.

In the version that Aera Vault uses only the "restrict LP" field can be overwritten and by carefully crafting the value of `endTime`, the "restrict LP" boolean can be switched off, allowing anyone to use `joinPool`.

The Manager could cause this behavior via the `updateWeightsGradually()` function while the Owner could do it via `enableTradingWithWeights()`.

Note: This issue has been reported to Balancer by the Spearbit team.

```
contract ManagedPool is BaseWeightedPool, ReentrancyGuard { // f14de92ac443d6daf1f3a42025b1ecdb8918f22e
    // [ 64 bits  | 119 bits |     1 bit     |  32 bits   |   32 bits  |     7 bits    |   1 bit   ]
    // [ reserved |  unused  | restrict LP | end time   | start time | total tokens | swap flag ]
    // |MSB

    function _startGradualWeightChange(uint256 startTime, uint256 endTime, ... ) ... {
        ...
        _setMiscData(
            _getMiscData().insertUint32(startTime, _START_TIME_OFFSET).insertUint32(endTime,
↪   _END_TIME_OFFSET)
        ); // this convert the values to 32 bits
        ...
    }
}
```

In the latest version of `ManagedPool` many more fields can be overwritten, including:

- LP flag
- Fee end/Fee start
- Swap flag

```
contract ManagedPool is BaseWeightedPool, AumProtocolFeeCache, ReentrancyGuard { // current version
    // [ 64 bits   |  1 bit  | 31 bits |   1 bit   |  31 bits   |  64 bits |  32 bits |  32 bits  ]
    // [ swap fee | LP flag | fee end | swap flag | fee start | end swap | end wgt  | start wgt ]
    // |MSB                                                                                  LSB|
```

The following POC shows how fields can be manipulated.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.13;
import "hardhat/console.sol";

contract checkbalancer {
    uint256 private constant _MASK_1 = 2**(1) - 1;
    uint256 private constant _MASK_31 = 2**(31) - 1;
    uint256 private constant _MASK_32 = 2**(32) - 1;
    uint256 private constant _MASK_64 = 2**(64) - 1;
    uint256 private constant _MASK_192 = 2**(192) - 1;
    // [ 64 bits   |  1 bit   | 31 bits |    1 bit    |  31 bits   |  64 bits |  32 bits |  32 bits   ]
    // [ swap fee | LP flag | fee end | swap flag | fee start | end swap | end wgt  | start wgt ]
    // |MSB                                                                                      LSB|
    uint256 private constant _WEIGHT_START_TIME_OFFSET = 0;
    uint256 private constant _WEIGHT_END_TIME_OFFSET = 32;
    uint256 private constant _END_SWAP_FEE_PERCENTAGE_OFFSET = 64;
    uint256 private constant _FEE_START_TIME_OFFSET = 128;
    uint256 private constant _SWAP_ENABLED_OFFSET = 159;
    uint256 private constant _FEE_END_TIME_OFFSET = 160;
    uint256 private constant _MUST_ALLOWLIST_LPS_OFFSET = 191;
    uint256 private constant _SWAP_FEE_PERCENTAGE_OFFSET = 192;

    function insertUint32(bytes32 word,uint256 value,uint256 offset) internal pure returns (bytes32) {
        bytes32 clearedWord = bytes32(uint256(word) & ~(_MASK_32 << offset));
        return clearedWord | bytes32(value << offset);
    }
    function decodeUint31(bytes32 word, uint256 offset) internal pure returns (uint256) {
        return uint256(word >> offset) & _MASK_31;
    }
    function decodeUint32(bytes32 word, uint256 offset) internal pure returns (uint256) {
        return uint256(word >> offset) & _MASK_32;
    }
    function decodeUint64(bytes32 word, uint256 offset) internal pure returns (uint256) {
        return uint256(word >> offset) & _MASK_64;
    }
    function decodeBool(bytes32 word, uint256 offset) internal pure returns (bool) {
        return (uint256(word >> offset) & _MASK_1) == 1;
    }
    function insertBits192(bytes32 word,bytes32 value,uint256 offset) internal pure returns (bytes32) {
        bytes32 clearedWord = bytes32(uint256(word) & ~(_MASK_192 << offset));
        return clearedWord | bytes32((uint256(value) & _MASK_192) << offset);
    }

constructor() {
    bytes32 poolState;
    bytes32 miscData;
    uint startTime = 1 + 2*2**32;
    uint endTime = 3 + 4*2**32 + 5*2**(32+64) + 2**(32+64+31) + 6*2**(32+64+31+1) + 2**(32+64+31+1+31)
↪   + 7*2**(32+64+31+1+31+1);

    poolState = insertUint32(poolState,startTime, _WEIGHT_START_TIME_OFFSET);
    poolState = insertUint32(poolState,endTime, _WEIGHT_END_TIME_OFFSET);
    miscData  = insertBits192(miscData,poolState,0);

    console.log("startTime",            decodeUint32(miscData,  _WEIGHT_START_TIME_OFFSET)); // 1
    console.log("endTime",              decodeUint32(miscData,  _WEIGHT_END_TIME_OFFSET));   // 3
    console.log("endSwapFeePercentage", decodeUint64(miscData,  _END_SWAP_FEE_PERCENTAGE_OFFSET)); // 4
    console.log("Fee startTime",        decodeUint31(miscData,  _FEE_START_TIME_OFFSET)); // 5
    console.log("Swap enabled",         decodeBool(miscData,    _SWAP_ENABLED_OFFSET)); // true
    console.log("Fee endTime",          decodeUint31(miscData,  _FEE_END_TIME_OFFSET)); // 6
    console.log("AllowlistLP",          decodeBool(miscData,    _MUST_ALLOWLIST_LPS_OFFSET)); // true
    console.log("Swap fee percentage",  decodeUint64(poolState, _SWAP_FEE_PERCENTAGE_OFFSET)); // 7
    console.log("Swap fee percentage",  decodeUint64(miscData,  _SWAP_FEE_PERCENTAGE_OFFSET)); // 0 due
↪   to miscData conversion
}
}
```

Recommendation: Make use of a `ManagedPool.sol` version which solves this issue.

In the meantime, before any call is made to `pool.updateWeightsGradually()` verify that :

- `startTime<= type(uint32).max`

- `endTime <= type(uint32).max`

Gauntlet: Recommendation implemented in PR #145

Spearbit: Acknowledged. Recommendation has been implemented.

5.1.2 `sweep` function should prevent Treasury from withdrawing pool's BPTs

Severity: *Critical Risk*

Context: MammonVaultV1.sol#L559-L561

Description: The current `sweep()` implementation allows the `vault` owner (the `Treasury`) to sweep any token owned by the vault including BPTs (Balancer Pool Tokens) that have been minted by the Vault during the pool's `initialDeposit()` function call.

The current vault implementation does not need those BPTs to withdraw funds because they are passed directly through the `AssetManager` flow via `withdraw()/finalize()`.

Being able to withdraw BPTs would allow the Treasury to:

- Withdraw funds without respecting the time period between `initiateFinalization()` and `finalize()` calls.
- Withdraw funds without respecting `Validator allowance()` limits.
- Withdraw funds without paying the manager's fee for the last `withdraw()`.
- `finalize` the pool, withdrawing all funds and selling valueless BPTs on the market.
- Sell or rent out BPTs and `withdraw()` funds afterwards, thus doubling the funds.

Swap fees would not be paid because `Treasury` could call `setManager(newManager)`, where the new manager is someone controlled by the Treasury, subsequently calling `setSwapFee(0)` to remove the swap fee, which would be applied during an `exitPool()` event.

Note: Once the BPT is retrieved it can also be used to call `exitPool()`, as the `mustAllowlistLPs` check is ignored in `exitPool()`.

Recommendation: Add a check on the `token` input parameter to prevent Treasury from withdrawing the Pool's BTP tokens.

Gauntlet: Fixed in PR #132

Spearbit: Acknowledged.

5.2 High Risk

5.2.1 Manager can cause an immediate weight change

Severity: *High Risk*

Context: ManagedPool.sol#L254-L272, ManagedPool.sol#L620-L654, ManagedPool.sol#L680-L698

Description: Balancer's `ManagedPool` uses 32 bit values for `startTime` and `endTime` but it does not verify if those values exist within that range.

When `endTime` is set to `2**32` it becomes larger than `startTime` so the `_require(startTime <= endTime, ...)` statement will not revert. When `endTime` is converted to 32 bits it will get a value of `0`, so in `_calculateWeightChangeProgress()` the test `if (currentTime >= endTime) ...` will be true, causing the weight to immediately reach the end value.

This way the Manager can cause an immediate weight change via the `updateWeightsGradually()` function and open arbitrage opportunities.

Note: `startTime` is also subject to this overflow problem. Note: the same issues occur in the latest version of `ManagedPool`. Note: This issue has been reported to Balancer by the Spearbit team.

Also see the following issues:

- *Managed Pools are still undergoing development and may contain bugs and/or change significantly*
- *Important fields of Balancer can be overwritten by `EndTime`*

```
contract ManagedPool is BaseWeightedPool, ReentrancyGuard {
    function updateWeightsGradually(uint256 startTime, uint256 endTime, ... ) {
        ...
        uint256 currentTime = block.timestamp;
        startTime = Math.max(currentTime, startTime);
        _require(startTime <= endTime, Errors.GRADUAL_UPDATE_TIME_TRAVEL); // will not revert if
↪   endTime == 2**32
        ...
      _startGradualWeightChange(startTime, endTime, _getNormalizedWeights(), endWeights, tokens);
    }

    function _startGradualWeightChange(uint256 startTime, uint256 endTime, ... ) ... {
        ...
        _setMiscData(
            _getMiscData().insertUint32(startTime, _START_TIME_OFFSET).insertUint32(endTime,
↪   _END_TIME_OFFSET)
        ); // this convert the values to 32 bits
        ...
    }

    function _calculateWeightChangeProgress() private view returns (uint256) {
        uint256 currentTime = block.timestamp;
        bytes32 poolState = _getMiscData();

        uint256 startTime = poolState.decodeUint32(_START_TIME_OFFSET);
        uint256 endTime = poolState.decodeUint32(_END_TIME_OFFSET);

        if (currentTime >= endTime) {  // will be true if  endTime == (2**32) capped to 32 bits == 0
            return FixedPoint.ONE;
        } else  ...
        ...
    }
}
```

Recommendation: Make use of a `ManagedPool.sol` version which solves this issue.

In the meantime verify before `pool.updateWeightsGradually()` is called that:

- `startTime<= type(uint32).max`
- `endTime  <= type(uint32).max`

Gauntlet: Recommendation implemented in PR #145

Spearbit: Acknowledged.

5.2.2 `deposit` and `withdraw` functions are susceptible to sandwich attacks

Severity: *High Risk*

Context: MammonVaultV1.sol#L402-L453, MammonVaultV1.sol#L456-L514

Description: Transactions calling the `deposit()` function are susceptible to sandwich attacks where an attacker can extract value from deposits. A similar issue exists in the `withdraw()` function but the minimum check on the pool holdings limits the attack's impact.

Consider the following scenario (swap fees ignored for simplicity):

1. Suppose the Balancer pool contains two tokens, WETH and DAI, and weights are 0.5 and 0.5. Currently, there is 1 `WETH` and 3k `DAI` in the pool and `WETH` spot price is 3k.
2. The Treasury wants to add another 3k `DAI` into the Aera vault, so it calls the `deposit()` function.
3. The attacker front-runs the Treasury's transaction. They swap 3k `DAI` into the Balancer pool and get out 0.5 `WETH`. The weights remain 0.5 and 0.5, but because `WETH` and `DAI` balances become 0.5 and 6k, `WETH`'s spot price now becomes 12k.
4. Now, the Treasury's transaction adds 3k `DAI` into the Balancer pool and upgrades the weights to 0.5*1.5: 0.5 = 0.6: 0.4.
5. The attacker back-runs the transaction and swaps the 0.5 `WETH` they got in step 3 back to `DAI` (and recovers the `WETH`'s spot price to near but above 3k). According to the current weights, they can get 9k*(1 - 1/r) = 3.33k `DAI` from the pool, where r = (2ˆ0.4)ˆ(1/0.6).
6. As a result the attacker profits 3.33k - 3k = 0.33k `DAI`.

Recommendation: Potential mitigations include:

- Adopting a two-step deposit and withdraw model. First, disable trading and check that the pool's spot price is within range. If not, enable trading again and let arbitragers re-balance the pool. Once rebalanced, deposit or withdraw from the pool. Then enable trading again (possibly with weights).
- Avoid depositing or withdrawing if the pool balance has changed in the same block. The `lastChangeBlock` variable stores the last block number where the pool balance was modified. By ensuring `lastChangeBlock` is less than the current block number, same-block sandwich attacks can be prevented. Still, this mitigation does not avoid multi-block MEV attacks.
- Similar to slippage protection, add price boundaries as parameters to the `deposit()` and `withdraw()` functions to ensure pool's spot price is within boundaries before and after deposit or withdrawal. Revert the transaction if boundaries are not met.
- Use Flashbots to reduce sandwiching probabilities.

Gauntlet: As discussed, this is a problem with spot price agnostic depositing into an AMM. V2 will introduce oracle-informed spot price updates. We will take the following actions for V1:

- Advise treasuries against making large deposits
- For sensitive/larger deposits, offer an option to reject the transaction if balances have been changed in the block (`lastChangeBlock`), implemented in PR #138.
- Advise treasuries to use flash bots when possible

Spearbit: Actions taken on a procedural and not technical level.

5.2.3 `allowance()` doesn't limit `withdraw()`s

Severity: *High Risk*

Context: PermissiveWithdrawalValidator.sol#L17-L27, IWithdrawalValidator.sol, MammonVaultV1.sol#L456-L514

Description: The `allowance()` function is meant to limit withdraw amounts. However, `allowance()` can only read and not alter state because its visibility is set to `view`. Therefore, the `withdraw()` function can be called on demand until the entire Vault/Pool balance has been drained, rendering the `allowance()` function ineffective.

```
function withdraw(uint256[] calldata amounts) ... {
    ...
    uint256[] memory allowances = validator.allowance();
    ...
    for (uint256 i = 0; i < tokens.length; i++) {
        if (amounts[i] > holdings[i] || amounts[i] > allowances[i]) {
            revert Mammon__AmountExceedAvailable(... );
        }
    }
}
```

```
// can't update state due to view
function allowance() external view override returns (uint256[] memory amounts) {
       amounts = new uint256[](count);
       for (uint256 i = 0; i < count; i++) {
           amounts[i] = ANY_AMOUNT;
       }
   }
```

Recommendation: Remove the `view` keyword from the `allowance()` template, e.g. from both `IWithdrawal-Validator.sol` and `PermissiveWithdrawalValidator.sol` to be able to update state in future versions of `al-lowance()`.

Gauntlet: I would say we need an additional callback to the Validator to notify it of actual withdraw amounts. In case when `allowance` is greater than `holdings` there is no way for the Validator to know how much of its allowance was actually used.

5.2.4 Managed Pools are still undergoing development and may contain bugs and/or significant changes

Severity: *High Risk*

Context: balancer-v2-monorepo

Description: The `ManagedPool` smart pool implementation of `WeightedPool` is still in active development by the Balancer team and could have unknown bugs. Additionally, the current version in balancer's github is different from the version used in Mannon Vault.

Note: The Gauntlet team has also flagged this as an issue.

Recommendation Use the latest version of `ManagedPool`. Consider using new functionalities like AUM (managementAumFeePercentage) fees.

Also see the following issues:

- *Check with Balancer team about best approach to add and remove funds*
- *Manager can cause an immediate weight change*
- *Important fields of Balancer can be overwritten by `EndTime`*

Only deploy once `ManagedPool` is stable and considered production ready.

Gauntlet: Fix implemented in PR #11. We should still update to the latest version at this stage (and continue to do so).

Spearbit: Acknowledged.

5.2.5 Malicious manager could cause Vault funds to be inaccessible

Severity: *High Risk*

Context: MammonVaultV1.sol#L794-L822

Description: The `calculateAndDistributeManagerFees()` function pushes tokens to the manager and if for unknown reasons this action fails the entire Vault would be blocked and funds become inaccessible. This occurs because the following functions depend on the execution of `calculateAndDistributeManagerFees()`:

`deposit()`, `withdraw()`, `setManager()`, `claimManagerFees()`, `initiateFinalization()`, and therefore `finalize()` as well.

Within `calculateAndDistributeManagerFees()` the function `safeTransfer()` is the riskiest and could fail under the following situations:

- A token with a callback is used, for example an ERC777 token, and the callback is not implemented correctly.
- A token with a blacklist option is used and the manager is blacklisted. For example USDC has such blacklist functionality. Because the manager can be an unknown party, a small risk exist that he is malicious and his address could be blacklisted in USDC.

Note: set as high risk because although probability is very small, impact results in Vault funds to become inaccessible.

```
function calculateAndDistributeManagerFees() internal {
    ...
    for (uint256 i = 0; i < amounts.length; i++) {
        tokens[i].safeTransfer(manager, amounts[i]);
    }
}
```

Recommendation: Beware of including tokens with callbacks such as ERC777 tokens into the Vault. Additionally, use a pull over push pattern to let the manager retrieve fees. This way the Vault can never get blocked.

This recommendation can be implemented as follows:

- Rename `calculateAndDistributeManagerFees()` to `calculateManagerFees()`.
- In `calculateManagerFees()` add up all management fees (for each manager address separate (e.g. in a mapping), to prevent having to push the fees in `setManager()` + keep track of total of the fees (`managersFeeTotal`) to make sure fees are not withdrawn);
- In `withdraw()` and `returnFunds()` make sure unclaimed fees cannot be withdrawn.
- Let the manager retrieve fees via `claimManagerFees()`, use `msg.sender` as index to the mapping with the fees. This function should retrieve Balancer funds, e.g. use the code of the second half of function `calculateAndDistributeManagerFees()`. It should also lower `managersFeeTotal` and the fee for `msg.sender`.

This also alleviates rounding issues with fees and reduces gas used by `deposit()` which could be relevant when pools are deployed with a large number of tokens.

An alternative could be to use `try/catch` around the call to `safeTransfer()`, but this way the fees aren't distributed properly.

Gauntlet: We will be incorporating the suggestions.

5.2.6 `updateWeightsGradually` allows change rates to start in the past with a very high `maximumRatio`

Severity: *High Risk*

Context: MammonVaultV1.sol#L599-L639

Description: The current `updateWeightsGradually` is using `startTime` instead of the minimal start time that should be `Math.max(block.timestamp, startTime)`. Because internally Balancer will use `startTime = Math.max(currentTime, startTime);` as the `startTime`, this allows to:

- Have a `startTime` in the past.
- Have a `targetWeights[i]` higher than allowed.

We also suggest adding another check to prevent `startTime > endTime`. Although Balancer replicates the same check it is still needed in the Aera implementation to prevent transactions to revert because of an underflow error on `uint256 duration = endTime - startTime;`

Recommendation: Update the code to correctly initialize the `startTime` value and add a check to prevent having `endTime` in the past (`startTime > endTime`). A possible solution looks as follows:

```
function updateWeightsGradually( ... ) ... {
+    startTime = Math.max(block.timestamp, startTime);
+    if ( startTime > endTime ) {
+        revert Mammon__WeightChangeEndBeforeStart();
+    }
    if (
-        Math.max(block.timestamp, startTime) +
+        startTime +
            MINIMUM_WEIGHT_CHANGE_DURATION >
        endTime
    ) {
        revert Mammon__WeightChangeDurationIsBelowMin(...)
            endTime - startTime,                        // no longer reverts
            MINIMUM_WEIGHT_CHANGE_DURATION
        );
    }
    ...
}
```

Gauntlet: Recommendation implemented in PR #146

Spearbit: Acknowledged.

5.2.7 The vault manager has unchecked power to create arbitrage using `setSwapFees`

Severity: *High Risk*

Context: MammonVaultV1.sol#L663-L679, BasePool.sol#L58-L59

Description: A previously known issue was that a malicious vault manager could arbitrage the vault like in the below scenario:

1. Set the swap fees to a high value by `setSwapFee` (10% is the maximum).
2. Wait for the market price to move against the spot price.
3. In the same transaction, reduce the swap fees to ~0 (0.0001% is the minimum) and arbitrage the vault.

The proposed fix was to limit the percentage change of the swap fee to a maximum of `MAXIMUM_SWAP_FEE_PERCENT_CHANGE` each time. However, because there is no restriction on how many times the `setSwapFee` function can be called in a block or transaction, a malicious manager can still call it multiple times in the same transaction and eventually set the swap fee to the value they want.

Recommendation: Enforce a cooldown period of reasonable length between two consecutive `setSwapFee` function calls.

5.2.8 Implement a function to claim liquidity mining rewards

Severity: *High Risk*

Context: MammonVaultV1.sol

Description: Balancer offers a liquidity mining rewards distribution for liquidity providers.

> Liquidity Mining distributions are available to claim weekly through the MerkleOrchard contract. Liquidity Providers can claim tokens from this contract by submitting claims to the tokens. These claims are checked against a Merkle root of the accrued token balances which are stored in a Merkle tree. Claiming through the MerkleOrchard is much more gas-efficient than the previous generation of claiming contracts, especially when claiming multiple weeks of rewards, and when claiming multiple tokens.

The `AeraVault` is itself the only liquidity provider of the Balancer pool deployed, so each week it's entitled to claim those rewards. Currently, those rewards cannot be claimed because the `AeraVault` is missing an implementation to interact with the MerkleOrchard contract, causing all rewards (BAL + other tokens) to remain in the MerkleOrchard forever.

Recommendation: Add a function to allow the vault owner (the Treasury) to claim those rewards. More information on how to claim rewards and interact with the contract can be found directly in the Balancer Documentation website.

Rewards claimed by the `AeraVault` can be lately distributed to the Treasury via the `sweep` function.

Gauntlet: Recommendation implemented in PR #146.

Spearbit: Acknowledged.

5.3 Medium Risk

5.3.1 Owner can circumvent `allowance()` via `enableTradingWithWeights()`

Severity: *Medium Risk*

Context: MammonVaultV1.sol#L564-L593

Description: The vault Owner can set arbitrary weights via `disableTrading()` and then call `enableTradingWithWeights()` to set the spot price and create arbitrage opportunities for himself. This way `allowance()` in `withdraw()` checks, which limit the amount of funds an owner can withdraw, can be circumvented.

Something similar can be done with `enableTradingRiskingArbitrage()` in combination with sufficient time.

Also see the following issues:

- `allowance()` doesn't limit `withdraw()`s
- `enableTradingWithWeights` allow the Treasury to change the pool's weights even if the swap is not disabled
- Separation of concerns Owner and Manager

```
function disableTrading() ... onlyOwnerOrManager ... {
    setSwapEnabled(false);
}

function enableTradingWithWeights(uint256[] calldata weights) ... onlyOwner ... {
    ...
    pool.updateWeightsGradually(timestamp, timestamp, weights);
    setSwapEnabled(true);
}

function enableTradingRiskingArbitrage() ... onlyOwner ... {
    setSwapEnabled(true);
}
```

Recommendation: Consider allowing only the manager to execute the `disableTrading()` function although this also has disadvantages. Additionally, use an oracle to determine spot price (as is already envisioned for the next versions of the protocol).

Gauntlet: For safety reasons we want the treasury to have full control over trading. Given our current trust model, this is won't be an issue for V1 so no action will be taken at this time.

Spearbit: Acknowledged.

5.3.2 Front-running attacks on `finalize` could affect received token amounts

Severity: *Medium Risk*

Context: MammonVaultV1.sol#L539, MammonVaultV1.sol#L899-L910

Description: The `returnFunds()` function (called by `finalize()`) withdraws the entire holdings in the Balancer pool but does not allow the caller to specify and enforce the minimum amount of received tokens. Without such check the `finalize()` function could be susceptible to a front-running attack.

A potential exploit scenario looks as follows:

1. The notice period has passed and the Treasury calls `finalize()` on the Aera vault. Assume the Balancer pool contains 1 `WETH` and 3000 `DAI`, and that `WETH` and `DAI` weights are both 0.5.
2. An attacker front-runs the Treasury's transaction and swaps in 3000 `DAI` to get 0.5 `WETH` from the pool.
3. As an unexpected result, the Treasury receives 0.5 `WETH` and 6000 `DAI`. Therefore an attacker can force the Treasury to accept the trade that they offer.

Although the Treasury can execute a reverse trade on another market to recover the token amount and distribution, not every Treasury can execute such trade (e.g., if a timelock controls it). Notice that the attacker may not profit from the swap because of slippage but they could be incentivized to perform such an attack if it causes considerable damage to the Treasury.

Recommendation: Possible mitigations include:

- Allowing the caller to specify the minimum amount of each token and revert the transaction if not enough tokens are available.
- Adopting a two-step finalization pattern. First, disable trading and check if the token amounts in the Balancer pool are as desired. If not, enable trading again and let arbitragers re-balance the pool. Once rebalanced, finalize the vault.
- Use Flashbots to reduce front-running probabilities.

Gauntlet: Based on our latest thinking, trading should be paused when `initiateFinalization` is run. That should resolve this issue.

Spearbit: `setSwapEnabled(false)` has been added in `initiateFinalization()` in PR #137. It is worth noting that pausing trading does not completely solve the issue. If `initiateFinalization()` happens to be front run (although not profitable for a frontrunner, it could still happen), then the token distributions could still be off. This situation should probably be detected (manually?) and corrected with `enableTradingWithWeights()` and `disableTrading()`.

Gauntlet: I think there are 2 things that are important:

- If the treasury is using withdraw and asking for a specific amount of tokens, that they don't get less than that. If there happens to be a front-running transaction, just like in an AMM they may not be able to withdraw what they want
- If the treasury is finalizing they should expect to retain a decent amount of the **value** of the pool, but since it's a liquidity share in an AMM, there aren't guarantees about the specific ratios of token amounts. The only guarantee is the relationship between token weights, balances and spot prices.

Spearbit: The value indeed stays the same. Only if the token distribution would be important you would want to solve this.

Assuming the token distribution doesn't matter then you might as well keep the code as is (unless there are other reasons to change).

[e.g frontrun of `initiateFinalization()` + trade pause has the same effect as frontrun of `finalize()` while trade hasn't been paused]

Gauntlet: I still like the proposal as we see other benefits in pausing trading. Since trading is primarily a means of rebalancing execution, we can shut it off post initiation of finalization to mitigate impermanent loss for the treasury.

Spearbit: Acknowledged. Beware that `enableTradingWithWeights()`, `enableTradingRiskingArbitrage()` and `disableTrading()` still work after `initiateFinalization()`. This could be put to good use but also unwanted (in that case additional checks are required in these functions).

5.3.3 `safeApprove` in `depositToken` could revert for non-standard token like `USDT`

Severity: *Medium Risk*

Context: MammonVaultV1.sol#L893

Description: Some non-standard tokens like `USDT` will revert when a contract or a user tries to approve an allowance when the spender allowance has already been set to a non zero value. In the current code we have not seen any real problem with this fact because the `amount` retrieved via `depositToken()` is approved send to the Balancer pool via `joinPool()` and `managePoolBalance()`. Balancer transfers the same amount, lowering the approval to 0 again. However, if the approval is not lowered to exactly 0 (due to a rounding error or another unforeseen situation) then the next approval in `depositToken()` will fail (assuming a token like `USDT` is used), blocking all further deposits.

Note: Set to medium risk because the probability of this happening is low but impact would be high.

We also should note that OpenZeppelin has officially **deprecated** the `safeApprove` function, suggesting to use instead `safeIncreaseAllowance` and `safeDecreaseAllowance`.

Recommendation: Adopt a safer approach to cover edge cases such as the abovementioned `USDT` token and implement the following solution:

```
function depositToken(IERC20 token, uint256 amount) internal {
    token.safeTransferFrom(owner(), address(this), amount);
-    token.safeApprove(address(bVault), amount);
+    uint256 allowance = token.allowance(address(this), address(bVault));
+    if (allowance > 0) {
+        token.safeDecreaseAllowance(address(bVault), allowance);
+    }
+    token.safeIncreaseAllowance(address(bVault), amount);
}
```

Please note that the `amount` that should be used as a parameter for `safeIncreaseAllowance` should follow the `recommendations` written in issue *Fee on transfer can block several functions*.

5.3.4 Consult with Balancer team about best approach to add and remove funds

Severity: *Medium Risk*

Context: MammonVaultV1.sol

Description: The Mammon Vault uses AssetManager's functionality of function `managePoolBalance()` to add and remove funds. The standard way to add and remove funds in Balancer is via `joinPool()` / `exitPool()`. Using the `managePoolBalance()` function might lead to future unexpected behavior. Additionally, this disables the capacity to implement the original intention of AssetManagers functionality, e.g. storing funds elsewhere to generate yield.

Recommendation: Doublecheck with the Balancer team which is the best approach to implement. If either ways are nonoptimal ask them to implement the functionality to support this.

In case the `joinPool()` / `exitPool()` path is recommended by the Balancer team, it can probably be implemented in the following way:

- Limit access to `joinPool()` via allowlist (as is already done)
- Limit access to `exitPool()` via a custom pool with an `onExit()` callback function (which could also integrate `allowance()`)
- Adjust the spotprice after `joinPool()` / `exitPool()` via `updateWeights()`.
- Perhaps use the AUM (managementAumFeePercentage) fees.
- Only keep the BPT (pool tokens) in the vault.

Gauntlet: Both ways are probably not the "intended" use case, current version seems a bit more elegant code-wise. We will get in touch with Balancer team about the best way to use these low-level functions.

Spearbit: Acknowledged.

5.3.5 Fee on transfer can block several functions

Severity: *Medium Risk*

Context: MammonVaultV1.sol#L456-L514

Description: Some tokens have a fee on transfer, for example USDT. Usually such fee is not enabled but could be re-enabled at any time. With this fee enabled the `withdrawFromPool()` function would receive slightly less tokens than the `amounts` requested from Balancer causing the next `safeTransfer()` call to fail because there are not enough tokens inside the contract. This means `withdraw()` calls will fail.

Functions `deposit()` and `calculateAndDistributeManagerFees()` can also fail because they have similar code.

Note: The function `returnFunds()` is more robust and can handle this problem. Note: The problem can be alleviated by sending additional tokens directly to the Aera Vault contract to compensate for fees, lowering the severity of the problem to medium.

```
function withdraw(uint256[] calldata amounts) ... {
    ...
    withdrawFromPool(amounts); // could get slightly less than amount with a fee on transfer
    ...
    for (uint256 i = 0; i < amounts.length; i++) {
        if (amounts[i] > 0) {
            tokens[i].safeTransfer(owner(), amounts[i]); // could revert it the full amounts[i] isn't
↪    available
            ...
        } ...
    }
}
```

Recommendation: Check the `balanceOf()` tokens before and after a `safeTransfer()` or `safeTransferFrom()`. Use the difference as the amount of tokens sent/received.

5.3.6 `enableTradingWithWeights` allow the Treasury to change the pool's weights even if the swap is not disabled

Severity: *Medium Risk*

Context: MammonVaultV1.sol#L574-L583

Description: `enableTradingWithWeights` is a function that can only be called by the owner of the `Aera Vault` contract and that should be used only to re-enable the `swap` feature on the pool while updating token weights.

The function does not verify if the pool's `swap` feature is enabled and for this reason, as a result, it allows the Treasury to act as the manager who is the only actor allowed to change the pool weights.

The function should add a check to ensure that it is only callable when the pool's `swap` is disabled.

Recommendation: Update the function to revert when the pool's swap is enabled.

```
function enableTradingWithWeights(uint256[] calldata weights)
    external
    override
    onlyOwner
    whenInitialized
{
+    bool isSwapEnabled = pool.getSwapEnabled();
+    if( isSwapEnabled ) {
+      revert Mammon__PoolSwapIsAlreadyEnabled();
+    }
    uint256 timestamp = block.timestamp;
    pool.updateWeightsGradually(timestamp, timestamp, weights);
    setSwapEnabled(true);
}
```

Gauntlet: Fixed in PR #126.

Spearbit: Acknowledged.

5.3.7 `AeraVault constructor` is not checking all the input parameters

Severity: *Medium Risk*

Context: MammonVaultV1.sol#L260-L345

Description: The `Aera Vault` constructor has the role to handle Balancer's `ManagedPool` deployment. The constructor should increase the number of user input validation and the `Gauntlet` team should be aware of the possible edge case that could happen given that the deployment of the `Aera Vault` is handled directly by the Treasury and not by the Gauntlet team itself.

We are going to list all the worst-case scenarios that could happen given the premise that the deployments are handled by the Treasury.

1. `factory` could be a wrapper contract that will deploy a `ManagedPool`. This would mean that the deployer could pass correct parameters to `Aera Vault` to pass these checks, but will use custom and malicious parameters on the `factory` wrapper to deploy the real Balancer pool.
2. `swapFeePercentage` value is not checked. On Balancer, the deployment will revert if the value is not inside this range `>= 1e12 (0.0001%) and <= 1e17 (10% - this fits in 64 bits)`. Without any check, the Gauntlet accept to follow the Balancer's swap requirements.
3. `manager_` is not checked. They could set the manager as the Treasury (owner of the vault) itself. This would give the Treasury the full power to manage the Vault. At least these values should be checked: `address(0)`, `address(this)` or `owner()`. The same checks should also be done in the `setManager()` function.
4. `validator_` could be set to a custom contract that will give full allowances to the Treasury. This would make the `withdraw()` act like `finalize()` allowing to withdraw all the funds from the vault/pool.

5. `noticePeriod_` has only a max value check. Gauntlet team explained that a time delay between the initialization of the finalize process and the actual `finalize` is needed to prevent the Treasury to be able to instantly withdraw all the funds. Not having a min value check allow the Treasury to set the value to `0` so there would be no delay between the `initiateFinalization()` and `finalize()` because `noticeTimeoutAt == block.timestamp`.
6. `managementFee_` has no minimum value check. This would allow the Treasury to not pay the manager because the `managerFeeIndex` would always be `0`.
7. `description_` can be empty. From the Specification PDF, the description of the vault has the role to "Describes vault purpose and modelling assumptions for differentiating between vaults". Being empty could lead to a bad UX for external services that needs to differentiate different vaults.

These are all the checks that are done directly by Balancer during deployment via the Pool Factory:

- `BasePool constructor#L94-L95` min and max number of tokens.
- `BasePool constructor#L102`token array is sorted following Balancer specification (sorted by token address).
- `BasePool constructor calling _setSwapFeePercentage` min and max value for `swapFeePercentage`.
- `BasePool constructor calling vault.registerTokens` token address uniqueness (can't have same token in the pool), it also checks that `token != IERC20(0)`. Following the path`BasePool` is calling `vault.registerTokens` that should call `function _registerMinimalSwapInfoPoolTokens` from `MinimalSwapInfoPoolsBalance`.
- ManagedPool constructor calling _startGradualWeightChange Check min value of weight and that the total sum of the weights are equal to 100%. `_startGradualWeightChange` internally check that `endWeight >= WeightedMath._MIN_WEIGHT` and `normalizedSum == FixedPoint.ONE`.

Recommendation:

- Create a factory to wrap both `AeraVault` and `Validator` deployment to reduce influence and possible malicious attack from external actors.
- Add a custom min/max value check for `swapFeePercentage` on top of Balancer's check if needed.
- Add checks on `manager_` value to prevent an empty manager (`address(0)`) or that the manager and`AeraVault` owner will be equal to the Treasury itself.
- Added a min value check to the `noticePeriod_` parameter if needed to prevent that the time between `initiateFinalization` and `finalize` call is too small.
- Add a min value check to the `managementFee_` parameter if needed to prevent the Treasury to not pay the manager.
- Add a check on `description_` to prevent to deploy a `AeraVault` with an empty description that would create confusion on web application that will display similar vaults.
- Check meticulously that future Balancer's version still maintain the same checks listed above. Consider replicating those checks during deployment to be future-proof.

We also recommend carefully documenting the possible consequences of supporting "special" types of tokens:

- Token with more than 18 decimals that are **not supported** by Balancer.
- Token with small number of decimals.
- ERC777 tokens.
- Token with fees on transfer.
- Token with blacklisting capabilities.

Gauntlet: In our trust model, we only decide to manage the vault if it has been correctly deployed. So leaving the focus to be on human error, I think the following are actionable:

3. (manager checks)

Add checks on manager_ value to prevent an empty manager (address(0)) or that the manager and `Aera Vault` owner will be equal to the Treasury itself

4. (description checks)

    Add a check on description_ to prevent to deploy a `Aera Vault` with an empty description that would create confusion on web application that will display similar vaults.

I'll also take a documentation action for these:

> Token with more than 18 decimals that are not supported by Balancer Token with small number of decimals ERC777 tokens Token with fees on transfer Token with blacklisting capabilities

5.3.8 Possible mismatch between `Validator.count` and `AeraVault` assets count

Severity: *Medium Risk*

Context: PermissiveWithdrawalValidator.sol#L13, MammonVaultV1.sol#L456-L514

Description: A weak connection between `WithdrawalValidator` and `Aera Vault` could lead to the inability of withdrawing from a Vault.

Consider the following scenario:

The Validator is deployed with a `tokenCount` < than `Vault.getTokens().length`. Inside the `withdraw()` function we reference the following code block:

```
uint256[] memory allowances = validator.allowance();
uint256[] memory weights = getNormalizedWeights();
uint256[] memory newWeights = new uint256[](tokens.length);

for (uint256 i = 0; i < tokens.length; i++) {
    if (amounts[i] > holdings[i] || amounts[i] > allowances[i]) {
        revert Mammon__AmountExceedAvailable(
            address(tokens[i]),
            amounts[i],
            holdings[i].min(allowances[i])
        );
    }
}
```

A scenario where `allowances.length < tokens.length` would cause this function to revert with an Index out of bounds error. The only way for the Treasury to withdraw funds would be via the `finalize()` method which has a time delay.

Recommendation: Ensure that when `Aera Vault` and `Validator` are deployed, `Validator.count` is the same as the number of assets managed by the vault.

A potential solution is to create a Factory contract that will deploy both the `Aera Vault` and the `Validator`. In such case remember to correctly set up the `Aera Vault` Ownable contract because the current deployer is also the Vault's owner and in the case of a Factory, the owner of the vault would be the Factory itself. This would cause all `onlyOwner` calls to revert.

Additionally, see the following issue: *Ensure integrity of deployment of vault.*

Gauntlet: I think we are ok with trusting the treasury to deploy the validator. The main reason being is that we don't see a simple way to parametrize validators at the moment and each treasury partner will be highly trusted early on so we'd be starting with permissive validators and then leaning on treasuries to suggest some custom validators first.

Checked on `allowances.length` added in PR #141.

Spearbit: Acknowledged.

5.4 Low Risk

5.4.1 Ensure vault's deployment integrity

Severity: *Low Risk*

Context: MammonVaultV1.sol

Description: The treasury could deploy on purpose or by accident a slightly different version of the contract and introduce bugs or backdoors. This might not be recognized by parties taking on Manager responsibilities (e.g. usually Gauntlet will be involved here).

Recommendation: Consider using a factory to deploy the contract(s), so only parameters can be set.

Gauntlet: We are ok with trusting the treasury to deploy the validator. The main reason being is that we don't see a simple way to parametrize validators at the moment and each treasury partner will be highly trusted early on so we'd be starting with permissive validators and then leaning on treasuries to suggest some custom validators first.

Spearbit: Acknowledged.

5.4.2 Frequent calling of `calculateAndDistributeManagerFees()` lowers fees

Severity: *Low Risk*

Context: MammonVaultV1.sol#L682-L690, IManagerAPI.sol#L24-L25

Description: Via `calculateAndDistributeManagerFees()` a percentage of the Pool is subtracted and sent to the Manager. If this function is called too frequently his fees will be lower.

For example:

- If he calls it twice, while both time getting 1%, he actually gets: 1% + 1% * (100% - 1%) = 1.99%
- If he waits longer until he has earned 2%, he actually gets: 2%, which is slightly more than 1.99%
- If called very frequently the fees go to 0 (especially taking in account the rounding down). However the gas cost would be very high.

The Manager can (accidentally) do this by calling `claimManagerFees()`. The Owner can (accidentally or on purpose (e.g. using 0 balance change)) do this by calling `deposit()`, `withdraw()` or `setManager()`.

Note: Rounding errors make this slightly worse.

Also see the following issue: *Possible rounding down of fees*

```
function claimManagerFees() ... {
        calculateAndDistributeManagerFees(); // get a percentage of the Pool
    }
```

Recommendation: Encourage managers to not claim fees too often. For example, Natspec comments regarding the `claimManagerFees()` function inside the `IManagerAPI.sol` contract document that the function should not be called frequently.

Because the Owner is end responsible for the Vault and the gas costs most likely outweigh the fees it is probably not worth taking actions.

Gauntlet: Added comment on `claimManagerFees()` in PR #138.

Spearbit: Acknowledged.

5.4.3 OpenZeppelin best practices

Severity: *Low Risk*

Context: MammonVaultV1.sol#L4-L11

Description: The Aera Vault uses OpenZeppelin release 4.3.2 which is copied into their github. The current release of OpenZeppelin is 4.6.0 and includes several updates and security fixes.

The copies of the OpenZeppelin files are also (manually) changed to adapt the import paths. This has the risk of making a mistake in the process.

```
import "./dependencies/openzeppelin/SafeERC20.sol";
import "./dependencies/openzeppelin/IERC20.sol";
import "./dependencies/openzeppelin/IERC165.sol";
import "./dependencies/openzeppelin/Ownable.sol";
import "./dependencies/openzeppelin/ReentrancyGuard.sol";
import "./dependencies/openzeppelin/Math.sol";
import "./dependencies/openzeppelin/SafeCast.sol";
import "./dependencies/openzeppelin/ERC165Checker.sol";
```

Recommendation: Use recent versions and consider OpenZeppelin best practices:

- Quite a lot of project seem to use NPM install which leaves the risk for a supply chain attack on NPM. Another way would be to retrieve it from from OpenZeppelin releases repository but this also leaves the risk for a github supply chain attack.
- Preferably don't change the contracts to prevent mistakes.
- OpenZeppelin has a way to alert projects of vulnerabilities before public disclosures.
- Monitor the updates to the releases.
- Update to a new release if relevant bugfixes are applied. With every large release make sure a recent version of the OpenZeppelin contracts is used (but preferably also somewhat battle tested).

5.4.4 Possible rounding down of fees

Severity: *Low Risk*

Context: MammonVaultV1.sol#L794-L822

Description: If certain token has a few decimals numbers then fees could be rounded down to 0, especially if time between `calculateAndDistributeManagerFees()` is relatively small. This also could slightly shift the spot price because the balance of one coin is lowered while the other remains still. With fewer decimals the situation worsens, e.g. Gemini USD GUSD has 2 decimals, therefore the problem occurs with a balance of 10_000 GUSD. Note: The rounding down is probably neglectable in most cases.

```
function calculateAndDistributeManagerFees() internal {
   ...
    for (uint256 i = 0; i < tokens.length; i++) {
        amounts[i] = (holdings[i] * managerFeeIndex) / ONE;   // could be rounded down to 0
    }
...
}
```

With 1 USDC in the vault and 2 hours between `calculateAndDistributeManagerFees()`, the fee for USDC is rounded down to 0. This behavior is demonstrated in the following POC:

```
import "hardhat/console.sol";
contract testcontract {
    uint256 constant ONE = 10**18;
    uint managementFee = 10**8;   // MAX_MANAGEMENT_FEE = 10**9;
    constructor()  {
        uint holdings = 1E6;  // 1 USDC
        uint delay = 2 hours;
        uint managerFeeIndex = delay * managementFee;
        uint amounts = (holdings * managerFeeIndex) / ONE;
        console.log("Fee",amounts);   // fee is 0
    }
}
```

Recommendation: Consider implementing one (or more) of the following suggestions:

1. Sum all the fees and only claim and transfer them on the initiative of the manager. This solves other issues such as *Malicious manager could result in inaccessible funds of the Vault*.
2. Accept the rounding issue and document it. Do verify that the impact on the Pool price and the management fees is indeed neglectable. Managers could also be reimbursed manually if necessary.
3. If the potential change in the Pool price cannot be ignored: update the weights like `withdraw()` is doing.
4. Round the manager fees up. Perhaps only when they are 0. However be aware of repeatedly calling `claimManagerFees()` which could then get additional fees, although this is probably less than the gas costs.
5. Don't list tokens with a low number of decimals.

Gauntlet: We like 1 and 5. We think with 1, we would need to adjust the `sweep` function to disallow claiming vault tokens.

Spearbit:

> *we would need to adjust the sweep function to disallow claiming vault tokens.*

We would suggest to only claim the tokens from the vault when the manager claims them (e.g. have as few claims as possible as this prevents rounding issues) That way you don't have to change `sweep`

Gauntlet: Makes sense. FYI - if you claim too early, then you lose a little bit of value.

5.4.5 Missing `nonReentrant` modifier on `initiateFinalization()`, `setManager()` and `claimManagerFees()` functions

Severity: *Low Risk*

Context: MammonVaultV1.sol#L517, MammonVaultV1.sol#L545, MammonVaultV1.sol#L682

Description: The `initiateFinalization()` function is missing a `nonReentrant` modifier while `calculateAndDistributeManagerFees()` executes external calls. Same goes for `setManager()` and `claimManagerFees()` functions.

Recommendation: Consider adding the `nonReentrant` modifier to functions that perform external calls.

Gauntlet: Solved in PR #112.

Spearbit: Acknowledged.

5.4.6 Potential division by 0

Severity: *Low Risk*

Context: MammonVaultV1.sol#L402-L514, MammonVaultV1.sol#L728-L749

Description: If the balance (e.g. `holdings[]`) of a token is `0` in `deposit()` then the dividing by `holdings[]` would cause a revert.

Note: Function `withdraw()` has similar code but when `holdings[]==0` its not possible to `withdraw()` anyway. Note: The current Mannon vault code will not allow the balances to be 0. Note: Although not used in the current code, in order to do a `deregisterTokens()`, Balancer requires the balance to be 0. Additionally, refer to the following Balancer documentation about the-vault#deregistertokens.

The worst case scenario is `deposit()` not working.

```
function deposit(uint256[] calldata amounts) ... {
    ...
    for (uint256 i = 0; i < amounts.length; i++) {
        if (amounts[i] > 0) {
            depositToken(tokens[i], amounts[i]);
            uint256 newBalance = holdings[i] + amounts[i];
            newWeights[i] = (weights[i] * newBalance) / holdings[i]; // would revert if holdings[i] == 0
        } ...
    ...
}
```

Similar divisions by 0 could occur in `getWeightChangeRatio()`. The function is called from `updateWeightsGradually()`. If this is due to `targetWeight` being 0, then it is the desired result. Current `weight` should not be 0 due balancer checks.

```
 function getWeightChangeRatio(uint256 weight, uint256 targetWeight) ... {
        return
            weight > targetWeight
                ? (ONE * weight) / targetWeight    // could revert if targetWeight == 0
                : (ONE * targetWeight) / weight;    // could revert if weight== 0
    }
```

Recommendation: Determine what should happen in the unlikely event that balances become `0` in `deposit()` and adapt the code if necessary. If it is relevant to return readable revert/error messages then the division by 0 situation could be intercepted and trigger a direct revert.

Gauntlet: The code should revert with `0` balances since having `0` at either side of the AMM pool breaks spot prices and the ability to offer trades. Balancer does not allow `0` weight so this would be ok.

5.4.7 Use `ManagedPoolFactory` instead of `BaseManagedPoolFactory` to deploy the Balancer pool

Severity: *Low Risk*

Context: MammonVaultV1.sol#L305-L321, MammonVaultV1Fork.ts#L219-L223

Description: Currently the `Aera Vault` is using `BaseManagedPoolFactory` as the factory to deploy the Balancer pool while Balancer's documentation recommends and encourages the usage of `ManagedPoolFactory`.

Quoting the doc inside the `BaseManagedPoolFactory`:

> This is a base factory designed to be called from other factories to deploy a ManagedPool with a particular controller/owner. It should NOT be used directly to deploy ManagedPools without controllers. ManagedPools controlled by EOAs would be very dangerous for LPs. There are no restrictions on what the managers can do, so a malicious manager could easily manipulate prices and drain the pool. In this design, other controller-specific factories will deploy a pool controller, then call this factory to deploy the pool, passing in the controller as the owner.

Recommendation: Use `ManagedPoolFactory` instead of `BaseManagedPoolFactory` to deploy the Balancer pool following Balancer's best practices.

Gauntlet: Solved in PR #141.

Spearbit: Acknowledged.

5.4.8 Adopt the two-step ownership transfer pattern

Severity: *Low Risk*

Context: MammonVaultV1.sol#L762-L764, Ownable.sol#L61-L64

Description: To prevent the `Aera vault` Owner, i.e. the Treasury, from calling `renounceOwnership()` and effectively breaking vault critical functions such as `withdraw()` and `finalize()`, the `renounceOwnership()` function is explicitly overridden to revert the transaction every time. However, the `transferOwnership()` function may also lead to the same issue if the ownership is transferred to an uncontrollable address because of human errors or attacks on the Treasury.

Recommendation: Adopt the two-step ownership transfer pattern: (1) the owner sets the new owner, and (2) the new owner accepts the ownership.

Gauntlet: Solved in PR #132.

Spearbit: Acknowledged.

5.4.9 Implement zero-address check for `manager_`

Severity: *Low Risk*

Context: MammonVaultV1.sol#L267

Description: Non-existent zero-address checks inside the constuctor for the `manager_` parameter. If `manager_` becomes a zero address then calls to `calculateAndDistributeManagerFees` will burn tokens (transfer them to `address(0)`).

Recommendation: Implement zero-address check for `manager_` user input.

Example:

```
require(manager_ != address(0), "manager_: zero address");
```

Gauntlet: Fix implemented in PR #101.

Spearbit: Acknowledged.

5.5 Gas Optimization

5.5.1 Simplify tracking of managerFeeIndex

Severity: *Gas Optimization*

Context: MammonVaultV1.sol#L769-L774, MammonVaultV1.sol#L794-L822, MammonVaultV1.sol#L70

Description: The `calculateAndDistributeManagerFees()` function uses `updateManagerFeeIndex()` to keep track of management fees. It keeps track of both `managerFeeIndex` and `lastFeeCheckpoint` in storage variables (e.g. costing SLOAD/SSTORE). However, because `managementFee` is immutable this can be simplified to one storage variable, saving gas and improving code legibility.

```
uint256 public immutable managementFee; // can't be changed

function calculateAndDistributeManagerFees() internal {
    updateManagerFeeIndex();
    ...
    if (managerFeeIndex == 0) {
        return;
    }
    ...
    //  use managerFeeIndex
    ...
    managerFeeIndex = 0;
    ...
}

function updateManagerFeeIndex() internal {
    managerFeeIndex +=
        (block.timestamp - lastFeeCheckpoint) *
            managementFee;
    lastFeeCheckpoint = block.timestamp.toUint64();
}
```

Recommendation: Consider changing the code as follows:

```
function calculateAndDistributeManagerFees() internal {
    if (block.timestamp <= lastFeeCheckpoint) {
        return;
    }
    if (managementFee == 0) {
        return;
    }
    uint managerFeeIndex = (block.timestamp - lastFeeCheckpoint) * managementFee;  // managerFeeIndex
↪   is a local variable now
    lastFeeCheckpoint = block.timestamp;
    ...
    //  use managerFeeIndex
}
```

Remove the `updateManagerFeeIndex()` function.

Spearbit: Partly solved in PR #145, with a few attention points:

- `managerFeeIndex` could be local variable to safe gas (no need to store it between calls).
- In case `managementFee` happens to be 0 then gas is wasted (updated the recommendation above).

5.5.2 Directly call `getTokensData()` from `returnFunds()`

Severity: *Gas Optimization*

Context: MammonVaultV1.sol#L899-L910, MammonVaultV1.sol#L700-L708, MammonVaultV1.sol#L728-L749

Description: The function `returnFunds()` calls `getHoldings()` and `getTokens()`. Both functions call `getTokensData()` thus waste gas unnecessarily.

```
function returnFunds() internal returns (uint256[] memory amounts) {
        uint256[] memory holdings = getHoldings();
        IERC20[] memory tokens = getTokens();
   ...
}

function getHoldings() public view override returns (uint256[] memory amounts) {
    (, amounts, ) = getTokensData();
}

function getTokens() public view override returns (IERC20[] memory tokens)  {
    (tokens, , ) = getTokensData();
}
```

Recommendation: Call `getTokensData()` directly from `returnFunds()`.

Gauntlet: Solved in PR #100.

Spearbit: Acknowledged.

5.5.3 Change `uint32` and `uint64` to `uint256`

Severity: *Gas Optimization*

Context: MammonVaultV1.sol#L63-L88, MammonVaultV1.sol#L769-L774

Description: The contract contains a few variables/constants that are smaller than `uint256`: `noticePeriod`, `noticeTimeoutAt` and `lastFeeCheckpoint`. This doesn't actually save gas because they are not part of a `struct` and still take up a storage slot. It even costs more gas because additional bits have to be stripped off. Additionally, there is a very small risk of `lastFeeCheckpoint` wrapping to `0` in the `updateManagerFeeIndex()` function. If that would happen, `managerFeeIndex` would get far too large and too many fees would be paid out.

Finally, using `int256` simplifies the code.

```
contract MammonVaultV1 is IMammonVaultV1, Ownable, ReentrancyGuard {
    ...
    uint32 public immutable noticePeriod;
    ...
    uint64 public noticeTimeoutAt;
    ...
    uint64 public lastFeeCheckpoint = type(uint64).max;
    ...
}
```

```
function updateManagerFeeIndex() internal {
    managerFeeIndex +=
        (block.timestamp - lastFeeCheckpoint) *    // could get large when lastFeeCheckpoint wraps
        managementFee;
    lastFeeCheckpoint = block.timestamp.toUint64();
}
```

Recommendation:

- Replace `uint32` and `uint64` with `uint256`.

- Remove the conversion functions `toUint64()`.
- Remove `import "./dependencies/openzeppelin/SafeCast.sol";`
- Remove `using SafeCast for uint256;`

5.5.4 Use `block.timestamp` directly instead of assigning it to a temporary variable.

Severity: *Gas Optimization*

Context: MammonVaultV1.sol#L883, MammonVaultV1.sol#L580

Description: It is preferable to use `block.timestamp` directly in your code instead of assigning it to a temporary variable as it only uses 2 gas.

Recommendation: Use `block.timestamp` directly.

Example:

```
-    uint256 timestamp = block.timestamp;
-    pool.updateWeightsGradually(timestamp, timestamp, weights);
+    pool.updateWeightsGradually(block.timestamp, block.timestamp, weights);
```

Gauntlet: Solved in PR #102.

Spearbit: Acknowledged.

5.5.5 Consider replacing `pool.getPoolId()` with `bytes32 public immutable poolId` to save gas and external calls

Severity: *Gas Optimization*

Context: MammonVaultV1.sol#L394, MammonVaultV1.sol#L723-L725, MammonVaultV1.sol#L738, MammonVaultV1.sol#L855

Description: The current implementation of `Aera Vault` always calls `pool.getPoolId()` or indirectly `getPoolId()` to retrieve the `ID` of the `immutable` state variable `pool` that has been declared at `constructor` time.

The `pool.getPoolId()` is a getter function defined in the Balancer `BasePool` contract:

```
function getPoolId() public view override returns (bytes32) {
    return _poolId;
}
```

Inside the same `BasePool` contract the `_poolId` is defined as `immutable` which means that after creating a pool it will never change. For this reason it is possible to apply the same logic inside the `Aera Vault` and use an `immutable` variable to avoiding external calls and save gas.

Recommendation: Consider the following suggestions:

- Initializing an `immutable` variable with the `poolId` value after deploying the Balancer pool.
- Deleting all the reference to `getPoolId()` and `pool.getPoolId()`.
- Replacing those reference with a direct read to the immutable `poolId`.

```
+    bytes32 private immutable poolId;
constructor( ... ) {
    ...
    pool = IBManagedPool(
        IBManagedPoolFactory(factory).create(
            IBManagedPoolFactory.NewPoolParams({ ... })
        )
    );
+    poolId = pool.getPoolId();
    ...
}
```

Gauntlet: Fix has been implemented in PR #123.

Spearbit: Acknowledged.

5.5.6 Save values in temporary variables

Severity: *Gas Optimization*

Context: MammonVaultV1.sol#L427, MammonVaultV1.sol#L382, MammonVaultV1.sol#L481, MammonVaultV1.sol#L620, MammonVaultV1.sol#L495, MammonVaultV1.sol#L906, MammonVaultV1.sol#L301, MammonVaultV1.sol#L652, MammonVaultV1.sol#L808. MammonVaultV1.sol#L816, MammonVaultV1.sol#L858, MammonVaultV1.sol#L876

Description: We observed multiple occurrences in the codebase where `<var>.length` was used in `for` loops. This could lead to more gas consumption as `.length` gets called repetitively until the `for` loop finishes.

When indexed variables are used multiple times inside the loop in a read only way these can be stored in a temporary variable to save some gas.

```
for (uint256 i = 0; i < tokens.length; i++) {  // tokens.length has to be calculated repeatedly
    ...
    ... = tokens[i].balanceOf(...);                        // tokens[i] has to be evaluated multiple times
    tokens[i].safeTransfer(owner(), ...);
}
```

Recommendation: The value of `<var>.length` could be saved in a temporary variable for saving some gas cost. Also `<var>[i]` could be saved in a temporary variable for saving some gas cost, provided it is used in a read only way.

Example:

```
uint256 length = tokens.length; //temporary variable
for (uint256 i = 0; i < length; i++) {
    ...
    IERC20 tmpToken = tokens[i];   //temporary variable
    ... = tmpToken.balanceOf(...);
    tmpToken.safeTransfer(owner(), ...);
}
```

Gauntlet: Solved in PR #100.

Spearbit: Acknowledged.

5.6 Informational

5.6.1 Aera could be prone to out-of-gas transaction revert when managing a high number of tokens

Severity: *Informational*

Context: MammonVaultV1.sol#L350-L399, MammonVaultV1.sol#L402-L453, MammonVaultV1.sol#L456-L514, MammonVaultV1.sol#L531-L541

Description: The Balancer `ManagedPool` used by Aera has a max limit of 50 token.

Functions like: `initialDeposit()`, `deposit()`, `withdraw()` and `finalize()` involve numerous external direct and indirect (made by Balancer itself when called by `Aera`) calls and math calculations that are done for each token managed by the pool.

The functions `deposit()` and `withdraw()` are especially gas intensive, given that they also internally call `calculateAndDistributeManagerFees()` that will transfer, for each token, a management fee to the manager.

For these reasons Aera should be aware that a high number of tokens managed by the `Aera Vault` could lead to out-of-gas reverts (max block size depends on which chain the project will be deployed).

Recommendation: Consider the following suggestions:

- Monitor and tests gas consumption from those functions, simulating a managed pool that handles the max number of tokens allowed by the Balancer's `ManagedPool`.
- Consider adding a custom max limit on the number of tokens supported by the `AeraPool`.
- Consider applying the suggestions reported in issue "Malicious manager could result in inaccessible funds of the Vault".
- Consider the max block size currently available for the chains on which the project will be deployed.

Gauntlet: We are starting with 2 tokens for V1 and only gradually increasing. We will be incorporating the suggestions in *Malicious manager could result in inaccessible funds of the Vault*.

5.6.2 Use a consistent way to call `getNormalizedWeights()`

Severity: *Informational*

Context: MammonVaultV1.sol

Description: The functions `deposit()` and `withdraw()` call function `getNormalizedWeights()` while the function `updateWeightsGradually()` and `cancelWeightUpdates()` call `pool.getNormalizedWeights()`. Although this is functionally the same, it is not consistent.

```
function deposit(uint256[] calldata amounts) ... {
  ...
   uint256[] memory weights = getNormalizedWeights();
   ...
   emit Deposit(amounts, getNormalizedWeights());
}

function withdraw(uint256[] calldata amounts) ... {
   ...
   uint256[] memory weights = getNormalizedWeights();
   ...
   emit Withdraw(amounts, allowances, getNormalizedWeights());
}

function updateWeightsGradually(...) ... {
   ...
   uint256[] memory weights = pool.getNormalizedWeights();
   ...
}

function cancelWeightUpdates() ... {
   ...
   uint256[] memory weights = pool.getNormalizedWeights();
   ...
}

function getNormalizedWeights() ... {
   return pool.getNormalizedWeights();
}
```

Recommendation: Consistently using either `getNormalizedWeights()` or `pool.getNormalizedWeights()`.

Gauntlet: Fixed in PR #137.

Spearbit: Acknowledged.

5.6.3 Add function `disableTrading()` to `IManagerAPI.sol`

Severity: *Informational*

Context: MammonVaultV1.sol#L347-L596, IManagerAPI.sol

Description: The `disableTrading()` function can also be called by managers because of the `onlyOwnerOrManager`modifier. However in `MammonVaultV1.sol` it is located in the `PROTOCOL API` section. It is also not present in `IManagerAPI.sol`.

```
contract MammonVaultV1 is IMammonVaultV1, Ownable, ReentrancyGuard {
    /// PROTOCOL API ///

    function disableTrading() ... onlyOwnerOrManager ... {
    ...
    }
    /// MANAGER API ///
}
```

```
interface IManagerAPI {
    function updateWeightsGradually(...) external;
    function cancelWeightUpdates() external;
    function setSwapFee(uint256 newSwapFee) external;
    function claimManagerFees() external;
} // disableTrading() isn't present
```

Recommendation: Update the comments in `MammonVaultV1.sol`. Add `disableTrading()` to `IManagerAPI.sol`.

5.6.4 Doublecheck layout functions

Severity: *Informational*

Context: MammonVaultV1.sol

Description: Different ways are used to layout functions. Especially the part between `(` ... `)` and between `)` ... `{` is sometimes done on one line and sometimes split in multiple lines. Also `{` is sometimes at the end of a line and sometimes at the beginning. Although the layout is not disturbing it might be useful to doublecheck it.

Here are a few examples of different layouts:

```
function updateWeights(uint256[] memory weights, uint256 weightSum)
    internal
{
    ...
}

function depositToken(IERC20 token, uint256 amount) internal {
    ...
}

function updatePoolBalance(
    uint256[] memory amounts,
    IBVault.PoolBalanceOpKind kind
    ) internal {
    ...
}

function updateWeights(uint256[] memory weights, uint256 weightSum)
    internal
{
    ...
}

function updateWeightsGradually(
    uint256[] calldata targetWeights,
    uint256 startTime,
    uint256 endTime
    ) external override onlyManager whenInitialized whenNotFinalizing {
    ...
}

function getWeightChangeRatio(uint256 weight, uint256 targetWeight)
    internal
    pure
    returns (uint256)
{
    ...
}
```

Recommendation: Double check the layout rules for functions.

Gauntlet: We use Prettier for code formatting and this issue is related to the tool itself. I believe the formatting here is consistent as lines get broken up when they get too long rather than arbitrarily.

Spearbit: Acknowledged.

5.6.5 Use Math library functions in a consistent way

Severity: *Informational*

Context: MammonVaultV1.sol#L24, MammonVaultV1.sol#L486, MammonVaultV1.sol#L605

Description: In the `MammonVaultV1` contract, the OZ's `Math` library functions are attached to the type `uint256`. The `min` function is used as a member function whereas the `max` function is used as a library function.

Recommendation: To maintain consistency use `Math.min` at L486 or change L605 into the `a.max(b)` format.

Note: `using Math for uint256;` can be removed if the `Math...` pattern is used.

Gauntlet: Fixed in PR #111 and PR #139.

Spearbit: Acknowledged.

5.6.6 Separation of concerns Owner and Manager

Severity: *Informational*

Context: MammonVaultV1.sol#L545-L556

Description: The Owner and Manager roles are separated on purpose. Role separation usually helps to improve quality.

However this separation can be broken if the Owner calls `setManager()`. This way the Owner can set the Manager to one of his own addresses, do Manager functions (for example `setSwapFee()`) and perhaps set it back to the Manager.

Note: as everything happens on chain these actions can be tracked.

```
function setManager(address newManager) external override onlyOwner {
        if (newManager == address(0)) {
            revert Mammon__ManagerIsZeroAddress();
        }

        if (initialized && noticeTimeoutAt == 0) {
            calculateAndDistributeManagerFees();
        }

        emit ManagerChanged(manager, newManager);
        manager = newManager;
    }
```

Recommendation: Depending on the usefulness of separation of concerns consider doing the following:

- Add `require(newManager != owner())` to the `setManager()` function to prevent accidentally setting the Owner as the Manager.

Note: as discussed this doesn't protect against deliberately creating a separate address by the Owner and using that.

- Have a third role to set the Manager, possibly managed by a multisig.

Gauntlet: We think it's important for the treasury to be able to direct and immediate control over the manager, so we are not able to remove that power for V1. In future versions, this trust model may change.

5.6.7 Add modifier `whenInitialized` to function `finalize()`

Severity: *Informational*

Context: MammonVaultV1.sol#L531-L541

Description: The function `finalize()` does not have the modifier `whenInitialized` while most other functions have this modifier. This does not create any real issues because the function contains the check `noticeTimeoutAt == 0` which can only be skipped after `initiateFinalization()`, and this function does have the `whenInitialized` modifier.

```
function finalize() external override nonReentrant onlyOwner { // no modifier whenInitialized
    if (noticeTimeoutAt == 0) {  // can only be set via initiateFinalization()
        revert Mammon__FinalizationNotInitialized();
    }
    ...
}
```

Recommendation: Add the `whenInitialized` modifier to function `finalize()`.

Gauntlet: Fixed in PR #111.

Spearbit: Acknowledged.

5.6.8 Document the use of `mustAllowlistLPs`

Severity: *Informational*

Context: MammonVaultV1.sol#L305-L323

Description: In the Mannon Vault it is important that no other accounts can use `joinPool()` on the balancer pool. If other accounts are able to call `joinPool()`, they would get Balancer Pool Tokens (BPT) which could rise in value once more funds are added to the pool.

Luckily this is prevented by the `mustAllowlistLPs` parameter in `NewPoolParams`. Readers could easily overlook this parameter.

```
pool = IBManagedPool(
            IBManagedPoolFactory(factory).create(
                IBManagedPoolFactory.NewPoolParams({
                    vault: IBVault(address(0)),
                    name: name,
                    symbol: symbol,
                    tokens: tokens,
                    normalizedWeights: weights,
                    assetManagers: managers,
                    swapFeePercentage: swapFeePercentage,
                    pauseWindowDuration: 0,
                    bufferPeriodDuration: 0,
                    owner: address(this),
                    swapEnabledOnStart: false,
                    mustAllowlistLPs: true,    // prevent other account to use joinPool
                    managementSwapFeePercentage: 0
                })
            )
        );
```

Recommendation: Add a comment showing the significance of the `mustAllowlistLPs` parameter.

5.6.9 `finalize` can be called multiple times

Severity: *Informational*

Context: MammonVaultV1.sol#L531-L541

Description: The `finalize` function can be called multiple time, leading to the possibility to waste gas for no reason and emitting again a conceptually wrong `Finalized` event. Currently, there's no check that will prevent to call the function multiple time and there is no explicit flag to allow external sources (web app, external contract) to know whether the `AeraVault` has been finalized or not.

Scenario: the `AeraVault` has already been finalized but the owner (that could be a contract and not a single EOA) is not aware of it. He calls `finalize` again and wastes gas because of the external calls in a loop done in `returnFunds` and emit an additional event `Finalized(owner(), [0, 0, ..., 0])` with an array of zeros in the `amounts` event parameter.

Recommendation: One simple change that could make the UX/DX much better would be to have a state variable to track if the `finalize` function have been called already.

```
/// STORAGE SLOT START ///

+/// @notice Indicates that the Vault has been finalized
+bool public finalized;

function finalize() external override nonReentrant onlyOwner {
    if (noticeTimeoutAt == 0) {
        revert Mammon__FinalizationNotInitialized();
    }
    if (noticeTimeoutAt > block.timestamp) {
        revert Mammon__NoticeTimeoutNotElapsed(noticeTimeoutAt);
    }

+    finalized = true;

    uint256[] memory amounts = returnFunds();
    emit Finalized(owner(), amounts);
}
```

With the new `finalized` boolean, there are two possible solutions to the problem:

- Soft approach: the `finalized == true` in addition with the fact that all the token balances in the Balancer pool are empty (equal to 0) is a consequence that the `AeraVault` have been already finalized and funds have been withdrawn. This combined information can be used to let an external website/contract know that there is no reason to call `finalize` again.
- Hard approach: add a `if (finalized) revert Mammon__AlreadyFinalized();` check after the present checks to revert in the case that the `finalize` function have been already called previously. Something important to keep in mind is that this would prevent the `owner` of the Vault to be able to withdraw funds in the case where something on the `Balancer` side have gone wrong (without reverting the transaction) and not all the remaining funds have been sent to the `AeraVault`.

Gauntlet: Let's go for the hard option. Implemented in PR #137.

Spearbit: Acknowledged. The "hard approach" has been implemented.

5.6.10 Consider updating `finalize` to have a more "clean" final state for the `AeraVault`/Balancer pool

Severity: *Informational*

Context: MammonVaultV1.sol#L531-L541

Description: This is just a suggestion and not an issue per se. The `finalize` function should ensure that the pool is in a finalized state for both a better UX and DX.

Currently, the `finalize` function is only withdrawing all the funds from the pool after a `noticePeriod` but is not ensuring that the `swap` have been disabled and that all the rewards, entitled to the Vault (owned by the Treasury), have been claimed.

Recommendation: Consider implementing the following actions to have, as a result, a more clean vault/pool clean state after executing `finalize`:

- claim remaining vault rewards and sweep them
- disable `swap` if not already disabled
- Consider speaking with Balancer to understand which is the best practice to follow when a pool is being "retired". For example, should the LP token minted during the pool initial deposit be burned even if the Treasury/Vault will not be able to withdraw them because of issue "Sweep function should prevent the Treasury to withdraw pool's BPT"?

5.6.11 `enableTradingWithWeights` is not emitting an event for pool's weight change

Severity: *Informational*

Context: MammonVaultV1.sol#L574-L583

Description: `enableTradingWithWeights` is both changing the pool's weight and enabling the `swap` feature, but it's only emitting the `swap` related event (done by calling `setSwapEnabled`).

Both of those operations should be correctly tracked via events to be monitored by external tools.

Recommendation: The best thing to do is to create a new event that would also allow specifically to track this delicate operation that is done by the Treasury, that usually would not have the ability to update the pool's weight.

```
+/// @notice Emitted when enableTradingWithWeights is called.
+/// @param time timestamp of updates.
+/// @param weights Target weights of tokens.
+event EnabledTradingWithWeights(
+    uint256 time,
+    uint256[] weights
+);


function enableTradingWithWeights(uint256[] calldata weights)
    external
    override
    onlyOwner
    whenInitialized
{
    uint256 timestamp = block.timestamp;
    pool.updateWeightsGradually(timestamp, timestamp, weights);
-    setSwapEnabled(true);
+   pool.setSwapEnabled(true);
+   emit EnabledTradingWithWeights(timestamp, weights);
}
```

Gauntlet: Fixed in PR #126.

Spearbit: Acknowledged.

5.6.12 Document Balancer checks

Severity: *Informational*

Context: MammonVaultV1.sol#L574-L583, ManagedPool.sol#L375-L387

Description: Balancer has a large number of internal checks. We've discussed the use of additional checks in the Aera Vault functions. The advantage of this is that it could result in more user friendly error messages. Additionally it protects against potential future change in the Balancer code.

```
contract MammonVaultV1 is IMammonVaultV1, Ownable, ReentrancyGuard {
    function enableTradingWithWeights(uint256[] calldata weights) ... {
    {
        ... // doesn't check weights.length
        pool.updateWeightsGradually(timestamp, timestamp, weights);
        ...
    }
```

Balancer code:

```
function updateWeightsGradually( ..., uint256[] memory endWeights) ... {
    (IERC20[] memory tokens, , ) = getVault().getPoolTokens(getPoolId());
    ...
    InputHelpers.ensureInputLengthMatch(tokens.length, endWeights.length); // length check is here
    ...
    }
```

Recommendation: Consider documenting where the code relies on checks in balancer. This makes reasoning about the correctness of the Aera Vault easier, for Aera developers, auditors and users of the code.

Gauntlet: We reasoned that since Balancer has named errors it should be clear enough when those invariants are broken. This is a treasury product which people will interact with mainly through the UI.

5.6.13 Rename `FinalizationInitialized` to `FinalizationInitiated` for code consistency

Severity: *Informational*

Context: MammonVaultV1.sol#L165, MammonVaultV1.sol#L207, MammonVaultV1.sol#L517

Description: The function at L517 was renamed from `initializeFinalization` to `initiateFinalization` to avoid confusion with the Aera vault initialization. For code consistency, the corresponding event and error names should be changed.

Recommendation: Rename the event at L165 to `FinalizationInitiated`, and the error at L207 to `Mammon__-FinalizationNotInitiated`.

Gauntlet: Fixed in PR #111.

Spearbit: Acknowledged.

5.6.14 Consider enforcing an explicit check on token order to avoid human error

Severity: *Informational*

Context:

- MammonVaultV1.sol#L350
- MammonVaultV1.sol#L402
- MammonVaultV1.sol#L456
- MammonVaultV1.sol#L574
- MammonVaultV1.sol#L599-L603

Description: The Balancer protocol require (and enforce during the pool creation) that the pool's token must be ordered by the token address.

The following functions accept an `uint256[]` of `amounts` or `weights` without knowing if the order inside that array follow the same order of the tokens inside the Balancer pool.

- `initialDeposit`
- `deposit`
- `withdraw`
- `enableTradingWithWeights`
- `updateWeightsGradually`

While it's impossible to totally prevent the human error (they could specify the correct token order but wrongly swap the input order of the amount/weight) we could force the user to be more aware of the specific order in which the amounts/weights must be specified.

A possible solution applied to the `initialDeposit` as an example could be:

```
function initialDeposit(IERC20[] calldata tokensSorted, uint256[] calldata amounts)
    external
    override
    onlyOwner
{
    // ... other code

    IERC20[] memory tokens = getTokens();

    // check that also the tokensSorted length match the lenght of other arrays
    if (tokens.length != amounts.length || tokens.length != tokensSorted.length) {
        revert Mammon__AmountLengthIsNotSame(
            tokens.length,
            amounts.length
        );
    }


    // ... other code

    for (uint256 i = 0; i < tokens.length; i++) {
        // check that the token position associated to the amount has the same position of the one in
↪   the balancer pool
        if( address(tokens[i]) != address(tokensSorted[i]) ) {
            revert Mammon__TokenOrderIsNotSame(
                address(tokens[i]),
                address(tokensSorted[i]),
                i
            );
        }
        depositToken(tokens[i], amounts[i]);
    }

    // ... other code
}
```

Another possible implementation would be to introduce a custom struct

```
struct TokenAmount {
  IERC20 token;
  uint256 value;
}
```

Update the function signature `function initialDeposit(TokenAmount[] calldata tokenWithAmount)` and update the example code following the new parameter model.

It's important to note that while this solution will not completely prevent the human error, it will increase the gas consumption of each function.

Recommendation: Evaluate the possibility to reduce hypothetical human errors, enforcing a more explicit check on token amounts/weights order when updating the pool at the cost of increasing the gas consumption.

Gauntlet: Spearbit:

5.6.15 Swap is not enabled after `initialDeposit` execution

Severity: *Informational*

Context: MammonVaultV1.sol#L350-L399

Description: In the current deployment flow of the `AeraVault` the Balancer pool is created (by the `constructor`) with `swapEnabledOnStart` set as `false`.

When the pool receives their initial funds via `initialDeposit` the pool has still the swap functionality disabled. It is not explicitly clear in the specification document and in the code when the swap functionality should be enabled.

If the protocol wants to enable the `swap` as soon as the funds are deposited in the pool, they should call, after `bVault.joinPool(...)`, `setSwapEnabled(true)` or `enableTradingWithWeights(uint256[] calldata weights)` in case the external spot price is not aligned (both functions will also trigger a `SetSwapEnabled` event)

Recommendation: Ensure to call `setSwapEnabled(true)` or `enableTradingWithWeights(uint256[] calldata weights)` at the end of `initialDeposit` if you intend to enable swapping after the pool has been funded.

Gauntlet: Suggestion has been implemented in PR #123.

Spearbit: Acknowledged. This behavior change should also be documented in both high-level documentation and natspec.

5.6.16 Remove commented code and replace input values with Balancer enum

Severity: *Informational*

Context: MammonVaultV1.sol#L371-L380

Description: Inside `initialDeposit` function, there is some commented code (used as example) that should be removed for clarity and future confusion. The `initUserData` should not use direct input values (`0` in this case) but use the correct Balancer's enum value to avoid any possible confusion.

Following the Balancer documentation

- Encoding userData
- JoinKind

The correct way to declare `initUserData` is using the `WeightedPoolUserData.JoinKind.INIT` enum value.

Recommendation:

- Remove the code example comments
- Update the code to user Balancer's enum value

Gauntlet: Fixed in PR #102.

Spearbit: Acknowledged.

5.6.17 The `Created` event is not including all the information used to deploy the Balancer pool and are missing `indexed` properties

Severity: *Informational*

Context: MammonVaultV1.sol#L95-L111, MammonVaultV1.sol#L334-L342

Description: The current `Created` event is defined as

```
event Created(
    address indexed factory,
    IERC20[] tokens,
    uint256[] weights,
    address manager,
    address validator,
    uint32 noticePeriod,
    string description
);
```

And is missing some of the information that are used to deploy the pool. To allow external tools to better monitor the deployment of the pools, it should be better to include all the information that have been used to deploy the pool on Balancer.

The following information is currently missing from the event definition:

- `name`
- `symbol`
- `managementFee`
- `swapFeePercentage`

The event could also define both `manager` and `validator` as `indexed` event parameters to allow external tools to filter those events by those values.

Recommendation: We recommend to:

- Change `manager` and `validator`as `indexed` event parameter
- Add the missing information to the `event`

```diff
event Created(
    address indexed factory,
+   string name,
+   string symbol,
    IERC20[] tokens,
    uint256[] weights,
+   uint256 swapFeePercentage,
-   address manager,
-   address validator,
+   address indexed manager,
+   address indexed validator,
    uint32 noticePeriod,
+   uint256 managementFee,
    string description
);
```

- Update the `event` natspec comment to include documentation for the added event parameters
- Update the `emit Created` event code to include the missing informations

```
emit Created(
    factory,
+   name,
+   symbol,
    tokens,
    weights,
+   swapFeePercentage,
    manager_,
    validator_,
    noticePeriod_,
+   managementFee_
    description_
);
```

Gauntlet: Fixed in PR #101.

Spearbit: Acknowledged.

5.6.18 Rename temp variable `managers` to `assetManagers` to avoid confusions and any potential future mistakes

Severity: *Informational*

Context: MammonVaultV1.sol#L300

Description: The `managers` declared in the linked code (see context) are in reality Asset Manager that have a totally different role compared to the `AeraVault` Manager role. The AssetManager is able to control the pool's balance, withdrawing from it or depositing into it.

To avoid confusion and any potential future mistakes, it should be better to rename the temporary variable `managers` to a more appropriate name like `assetManagers`.

```
- address[] memory managers = new address[](tokens.length);
+ address[] memory assetManagers = new address[](tokens.length);

for (uint256 i = 0; i < tokens.length; i++) {
-     managers[i] = address(this);
+     assetManagers[i] = address(this);
}

pool = IBManagedPool(
    IBManagedPoolFactory(factory).create(
        IBManagedPoolFactory.NewPoolParams({
            vault: IBVault(address(0)),
            name: name,
            symbol: symbol,
            tokens: tokens,
            normalizedWeights: weights,
-           assetManagers: managers,
+           assetManagers: assetManagers,
            swapFeePercentage: swapFeePercentage,
            pauseWindowDuration: 0,
            bufferPeriodDuration: 0,
            owner: address(this),
            swapEnabledOnStart: false,
            mustAllowlistLPs: true,
            managementSwapFeePercentage: 0
        })
    )
);
```

Recommendation: Rename `managers` to `assetManagers` to avoid confusion and any potential future mistakes.

Gauntlet: Fixed in PR #123.

Spearbit: Acknowledged.

5.6.19 Move `description` declaration inside the storage slot code block

Severity: *Informational*

Context: MammonVaultV1.sol#L72-L74

Description: In the current code, the `description` state variable is in the block of `/// STORAGE ///` where all the `immutable` variable are re-grouped.

As the dev comment say, `string cannot be immutable bytecode but only set in constructor` so it would be better to move it inside the `/// STORAGE SLOT START ///` block of variables that regroup all the non-constant and non-immutable state variables.

Recommendation: Move the `description` state variable inside the `/// STORAGE SLOT START ///` code part

Gauntlet: Fixed in PR #112.

Spearbit: Acknowledged.

5.6.20 Remove unused imports from code

Severity: *Informational*

Context: MammonVaultV1.sol#L6

Description: The current implementation of the `MammonVaultV1` contract is importing OpenZeppelin `IERC165` interface, but that interface is never used or references in the code.

Recommendation: Remove the import

```
-import "./dependencies/openzeppelin/IERC165.sol";
```

Gauntlet: Fixed in PR #102.

Spearbit: Acknowledged.

5.6.21 `shortfall` is repeated twice in IWithdrawalValidator natspec comments

Severity: *Informational*

Context: IWithdrawalValidator.sol#L7-L8

Description: The word `shortfall` is repeated twice in the natspec comment.

Recommendation: Remove one instance of the word following the used style guide (row max length)

Gauntlet: Fixed in PR #101.

Spearbit: Acknowledged.

5.6.22 Provide definition of `weights` & `managementFee_` in the NatSpec comment

Severity: *Informational*

Context: MammonVaultV1.sol#L251-L259

Description: The NatSpec Format is special form of comments to provide rich documentation for functions, return variables and more. We observed an occurrence where the NatSpec comments are missing for two of the user inputs (`weights` & `managementFee_`).

Recommendation: Provide proper definition of `weights` & `managementFee_` in the NatSpec format comment i.e. `@param`

Gauntlet: Fixed in PR #101.

Spearbit: Acknowledged.